A Cuckoo's Egg in the Malware Nest
On-the-fly Signature-less Malware Analysis, Detection, and Containment for Large Networks

Damiano Bolzoni[1], Christiaan Schade[1] and Sandro Etalle[1,2]

[1]University of Twente, The Netherlands
[2]Eindhoven Technical University, The Netherlands

Abstract

Avatar is a new architecture devised to perform on-the-fly malware analysis and containment on ordinary hosts; that is, on hosts with no special setup. The idea behind Avatar is to inject the suspected malware with a specially crafted piece of software at the moment that it tries to download an executable. The special software can cooperate with a remote analysis engine to determine the main characteristics of the suspected malware, and choose an appropriate containment strategy, which may include process termination, in case the process under analysis turns out to be malicious, or let it continue otherwise. Augmented with additional detection heuristics we present in the paper, Avatar can also perform signature-less malware detection and containment.

Keywords: system security, malware detection and containment

1 Introduction

In the last half-decade, malware has evolved from a "hobby" for bored programmers to a business for cyber-criminals, who infect computer systems on a large scale to carry out illegal activities [20]. *Botnets* are a typical example of such business, and can be exploited to collect financial/sensitive user information. As noticed by Kolbitsch et al. [13] "malicious code, or malware, is one of the most pressing security problems on the Internet". Malware containment has thus become an urgent concern. Recent events, such as the RSA breach back in March 2011 [17], have shown that serious attackers employ *ad hoc* malware in multi-stage attacks to penetrate corporate networks and get hold of business-critical information.

Successful malware containment is based on two activities: *Detection* and *analysis.*

Detection. Concerning detection, the standard mechanisms employed against malware are based on signatures. Antivirus software and intrusion detection systems (both host- and network-based) rely on some sort of byte-matching techniques (either pattern- or hash-based) to detect the presence of malicious programs. To evade signature-based detection, malware writers can and do obfuscate the code using e.g., polymorphism, packing, encryption [4]. The result of the massive application of evasion techniques is that in the past few years the number of unique malware samples, and relative signatures, has increased dramatically. In Section 4 we discuss in more detail some of the latest results in signature-based malware detection.

Analysis. To understand how malware works, and to improve the crafting of detection signatures, researchers have developed several frameworks for automating dynamic malware analysis (e.g., Anubis [1], CWSandbox [7], Malheur [14]). These tools monitor the behaviour of a malware sample which is being executed in a severely controlled environment, and produce a detailed report of the operations it carries out (e.g., access/modifications to files, network activities, process execution, etc.).

Dynamic malware analysis is undoubtedly effective; however, it requires a specific analysis environment, which cannot be just *any* computer. Moreover – as almost all security techniques – it is not infallible: Among the possible evasion techniques, it is becoming a common practice for malware to check whether the execution takes place in a virtualized environment, which likely indicates the executable is being monitored [2]. Secondly, as reported by Comparetti et al. [6], some malicious behaviors, such as the so called "dormant functionalities", may remain long unobserved, for instance when they depend on circumstances which are hard to guess and to replicate dynamically.

Summarizing, current detection and analysis approaches suffer from the following limitations:

- (Existing) dynamic malware analysis approaches can only perform post-mortem, or offline, analysis of the malware sample, once it has been collected and submitted: Hence they lack the execution context information; moreover, they require specific setups.
- Detection and containment are based on signatures or behavioral models, and are therefore effective only for those samples for which an appropriate signature/model has been developed.
- The most effective approaches rely on the presence of an agent on the end-host to monitor system activities; such extra software component is invasive, might affect system performance, and cause additional burden because system administrators must plan carefully its development and maintenance.

In particular, security analysts do not get the chance to analyze and contain on-the-fly suspicious programs.

One would like to have in addition to standard tools a first line of defense against malware that does not require special settings for the host, nor predeployed signatures. Similarly to what happens with intrusion detection systems, and especially for large corporations, one could think of a security operation center (SOC), where security analysts are able to inspect on-going suspicious behaviours. Thus, automatic analysis tool could be employed to "select" suspicious programs for analysis, which would be then carried out with a mix of automatic and manual inspections.

Contribution In this paper, we present a novel approach to perform *on-the-fly* malware analysis and containment for large networks, without having to deploy any end-host component beforehand. Our architecture, we call it Avatar, relies on the observation that malware distribution is usually done in at least two phases: First the computer is infected with a tiny "spore", then in following phases this spore downloads one or more additional components from, for instance, some earlier compromised web servers. Those components, or "eggs", are used to extend the malware capabilities, e.g., hooking system APIs to grab user passwords, and usually come in the form of executables, or dynamic libraries. By doing so, malware writers can more easily avoid detection.

Our approach is based on the injection of "goodware" in the suspected malware: In the moment that the alleged malware attempts to dowload an egg, we substitute the egg with the goodware, we call it the *cuckoo's egg*[1]. This is an executable that – among other things – can carry out preliminary malware analysis, can terminate the malware or it can simply give the control back to the egg if the suspected malware turns out to be a legitimate program[2]. The current implementation of Avatar is meant to monitor Windows-based systems.

This is done without any special setup in the host

[1]Similarly to the cuckoos that engage in brood parasitism, our goodware is expected to circumvent the malware and take advantage of it for performing the analysis

[2]In some cases it may be illegal to inject in an application software other than the one meant to be downloaded. Avatar is meant to be deployed in corporate networks, where system and network administrators are (usually) allowed to monitor, and limit, users' actions.

that contains the suspected malware, which may be just any computer running any Windows operating system. Indeed, the cuckoo's egg can be generated and inoculated from the firewall, and the analysis can be done on a remote analysis engine to which the cuckoo's egg communicates after it has been injected in the host under analysis.

Our experiments show that this is all possible, and that the cuckoo's egg can, for instance, be designed to inspect the process that executes it after the download, or to send to Avatar's remote analysis engine information regarding the process, such as path on the file system, file handlers, network/registry activities, or even the executable itself. Depending on the current user's permissions, the malware analysis engine can even "order" the cuckoo's egg code to suspend or terminate the process, effectively containing a possible larger infection.

An important side-issue is when should one start being suspicious about a given process. In other words, when should the system suspect that a spore is actually trying to download an egg. For our experiments we have developed a heuristic method which works as follows: Malware is usually programmed to use several different download servers, as servers are often offline/discontinued. In practice, the spore often fails a number of times before succeeding in downloading the egg. Thus, we take into consideration per-host failed TCP connections and failed HTTP requests to identify malware attempts of downloading. A number of failed HTTP requests is a good indication of the presence of malware. Our experiments show that this method is surprisingly effective. However, one can devise other heuristics which may be applicable in other contexts. It is outside of the scope of this paper to make an inventory of such methods.

To the best of our knowledge, this is the first approach which – without the installation of any additional plug-in before hand – allows one to:

- **(analysis)** carry out on-the-fly remote analysis of a suspicious program;
- **(containment)** suspend or terminate the suspicious program directly on the infected host;
- **(detection)** in combination with the heuristics for detecting suspicious downloads, it can identify suspicious malware processes which can be immediately analyzed and contained if required.

We should remark that this is done without using signatures of any kind. Therefore, this approach can be used to detect, analyze and contain also zero-day malware and malware for which there is no signature available yet. For example, one could even think of a "paranoid" mode, in which a cuckoo's egg is shipped for each download of executables regardless the rate of failed connections.

We show that Avatar is effective as a lightweight first line of defense against malware, also allowing to do malware containment on hosts with no specific pre-deployed tools (agent-less). This is a crucial requirement for system administrators of large networks, as it eases the burden required to install additional software to perform an accurate monitoring.

It is important to stress that this approach can be adapted to work with any protocol, in our embodiment we choose HTTP because it is widely used by malware writers. Of course, this approach has limitations, and can be countered to some extent. These aspects are discussed in Section 2.6.

2 Architecture

The architecture of Avatar consists of three main parts. The download detection engine (DDE) is responsible for detecting suspicious attempts to download software components. The Cuckoo's Egg Generator (CEG) is responsible for crafting the special analysis software that will be sent to requesting host. Finally, the Malware Analysis Engine (MAE) is responsible for analysing the information provided by the injected cuckoo's egg and possibly initiate some containment strategies. We now provide a detailed description of each component.

2.1 Download Detection Engine

The download detection engine (DDE) detects (failed) download attempts that *might* be due to malware activity. Strictly speaking, the functioning of the DDE is orthogonal to that of the analysis and

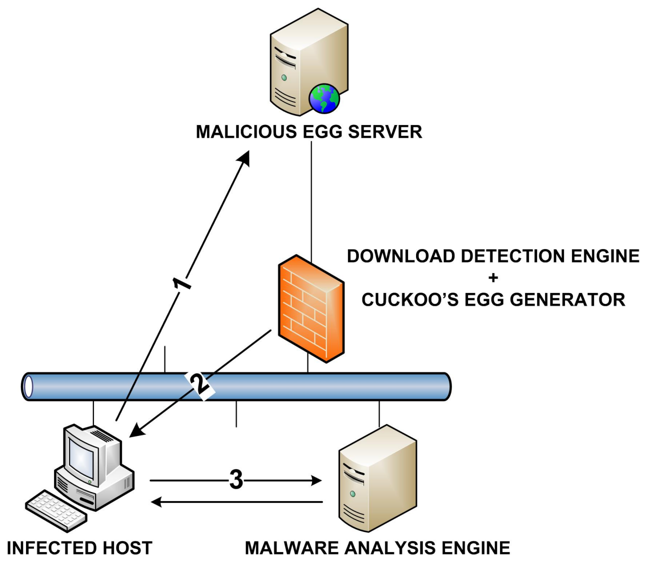

Figure 1: The Avatar architecture.

containment engines of Avatar, which on the other hand, are the core of the system. In fact, Avatar would work just as well also in combination with any other method one could devise to spot out suspicious download attempts. Nevertheless, it is easier to explain the whole architecture by starting from the DDE.

Our DDE is based on the fact that often malware fails a number of time to download eggs. This is due to the fact that download servers are often offline and/or taken down by security officers. In our embodiment the detection engine combines a modified version of the Threshold Random Walk (TRW) algorithm [10]. The engine builds a per-host model of normal usage, which takes into account the number of failed connections, and failed HTTP requests. In the case of malware, the former situation can occur when, e.g., the remote web server has been deactivated, the latter because the malicious content has been removed. As confirmed by our tests (see Section 3.1), these are not infrequent events. The resulting algorithm is simple, albeit effective, and could be easily expanded to include additional sources of information (e.g., DNS queries).

The DDE may be located at the network "border" with the Internet, in order to observe any outgoing connection and the data sent back by the remote host. As we said, while this component plays an important role in our approach, it is not the main driver of our idea. For instance, one could decide to inspect any executable download from the Internet, without the host having to failed a number of connections or HTTP requests before being flagged as suspicious.

The TRW algorithm is devised to detect scanning behaviours originating from a specific host in a monitored network. For each host a detection model is built. The outcome of a connection attempt is either "success" or "failure". After a number of observations of connection attempts for a certain host h, one would like to know if h is a scanner. To make such decision, a sequential hypothesis testing method is used. The basic premise is that there exists a distinct fixed ratio of failed and successful connections, and that this ratio is different when a host is a scanner. Furthermore, for each individual host this ratio value will eventually converge to some upper or lower boundaries, based on whether the host is a scanner or not.

We have adapted the TRW algorithm to take into account also successful and failed HTTP requests:

We currently employ only one model for both TCP connection and HTTP requests.

2.2 Cuckoo's Egg Generator (CEG)

Once the DDE identifies a suspicious download attempt, the CEG generates a specific executable/DLL to be fed back to the suspicious host. We now describe in details the purpose of the cuckoo's egg and its "internals".

2.2.1 The cuckoo's egg

The main goals of the cuckoo's egg are I) to gain as much knowledge as possible about the executing process, that, in case of malware, is usually the process that tried to download the "egg" and received the cuckoo's egg instead of it, and II) to take control over the parent process if necessary. The cuckoo's egg operates in two stages.

First, the cuckoo's egg "inspects" the execution environment. The reason for this is that different operating systems allow processes to execute certain operations with or without high privileges. Therefore, the cuckoo's egg may be allowed to perform only a restricted set of operations. For instance, beginning with Windows Vista, Microsoft includes a User Access Control (UAC) mechanism. The system can be set to notify the user when a process is about to modify some important system settings or execute potentially dangerous operations, so that the user can give explicit authorization. Because we want the cuckoo's egg to be as transparent as possible (for usability reasons), on Vista and later OSes, we cannot use a number of features, such as debugging mode, as these could (possibly) trigger the UAC.

The cuckoo's egg attempts to inject a specifically crafted DLL into its parent process with different access rights: The parent process can restrict the operation set the child process is allowed to perform. The different combinations of access right masks the cuckoo's egg uses are: PROCESS_ALL_ACCESS (highest privileges), TERMINATE_PROCESS | QUERY_INFO | READ, QUERY_INFO | READ and TERMINATE_PROCESS (lowest privileges).

Secondly, the injected DLL extracts, if allowed to, some information from the parent process (depending on the operational mode, see Section 2.4). This information includes: Full path, executable size on disk, DLLs that have been loaded, and information related to the current window attached to the process (if any), such as handle, size and caption text. At this stage, the cuckoo's egg's DLL attempts to determine quickly whether the parent process is malicious or not, and employs initially some heuristics based on the data above. Our experiments show that in most cases, one could tell straight after these heuristic checks whether the parent process is likely to be malware. For instance, a large executable size (more than 5 MB) is a sign of a non-malicious process: Malware writers tend to reduce the size of the "spore" to by-pass more easily anti-malware countermeasures. Similarly to [13], we also whitelist applications that could perform a licit download and later execute the downloaded file (e.g., Internet Explorer, Windows Update). Some limitations apply to these heuristics, and we discuss them in details in Section 2.6. An additional heuristic one might think to apply is the approach presented in [18], based on PE header analysis of suspicious programs.

If the heuristics do not indicate that the process is legitimate, then the information is passed to the MAE (discussed below) for remote analysis. Then – depending on the operational mode set and the user access rights – the cuckoo's egg can I) debug the parent process, II) let it run normally, III) "freeze" it, and, as a very last countermeasure, IV) terminate it.

In the first case, the cuckoo's egg can send back to the highly-instrumented malware analysis engine the debugged instructions. By doing so, we can "reply" on the remote analysis engine any operation and set whether we are debugging a malware process. However, our experiments show that this approach collects very little useful information on the parent process, as the malign process usually executes the egg(s) as the very last step of its run.

In the second and third cases, the cuckoo's egg sends back to Avatar the parent executable, and this is also the reason why we need to collect the parent's full path. By sending the whole executable, we can restart from scratch the process execution within our monitored environment and off-load a more accurate analysis.

2.2.2 Packaging the cuckoo's egg

Once the DDE notifies the CEG, the latter has to generate a suitable cuckoo's egg for the target and, depending on the operational mode, "attach" the original executable to the cuckoo's egg. We distinguish two cases, depending on whether the original executable is available, because it could be downloaded, or not.

As we mentioned earlier, the original executable requested by the host may not be available. In this case, only the cuckoo's egg is sent back to the target without any further processing. If, on the other hand, the originally requested executable is available, and the operational mode allows to do so, the CEG "forces" first the execution of the cuckoo's egg, and then of the "real" executable. Hence, the main concern when shipping the cuckoo's egg is to preserve the egg's functionalities as much as possible. There are several ways to achieve this, two of which are discussed here, each preserving the functionality in a different way:

- injecting a DLL loader stub through Portable Executable injection;

- shipping a replacement-executable that fetches and executes the egg after the parent process has been analyzed.

In the first case, the Portable Executable (PE) file header of the downloaded egg is altered. The PE format [15] is a file format for executables, object code, and DLLs, used by Windows since early NT versions. When an executable is launched, the system process loader uses the information included in the PE header to carry out operations such as: Filling in-memory data structures, loading required DLLs, and jumping to the entry point of the executable. In this case, the CEG appends the cuckoo's egg to the egg's executable file, next the egg's Entry Point is modified to point to a loader stub that will unpack the engine and write it to a file after which it will be loaded like any regular DLL. This method is rather complex, it presents the disadvantage that it might trigger the antivirus (unless some packing techniques are used) and requires the *LoadLibrary* and *GetProcAddress* offsets to be available in the egg's PE header, which is usually the case, though.

The second method is much simpler, requires no modifications to the egg's executable file, is usually not flagged as malicious by the antivirus and does not make any assumptions on the egg's PE header. A stand-alone cuckoo's egg is sent back and, once the analysis is over, it downloads and executes the "original" egg. The downside to this approach is that any relation that the egg may want to set up with its parent is lost. Moreover, this could significantly slow down the execution, by introducing an additional download latency.

2.3 Malware Analysis Engine

The MAE is the core component of the Avatar architecture. It is responsible for analysing the information sent by the cuckoo's egg. If necessary, it should run the suspected executable in a protected environment. From a functional point of view, it does not differ from other malware analysis tools. Once the sample to analyse is received, it is executed and any operation performed is recorded and logged. The execution report can be then dispatched to a security analyst, who can set a final verdict about the maliciousness of the sample, in case the executed program's nature remains unclear.

The MAE is also used to store information about whitelisted programs, which the cuckoo's egg will consider as non-suspicious. By doing so, we can basically centralize our architecture, making it possible to "update" crucial information about malware in one step.

2.4 Operational modes

As networks, and hosts, require different confidentiality and availability levels, users need to control the way the cuckoo's egg could affect the execution of processes. As in the case of all detection and prevention systems, false positives are always possible, so one has to find an appropriate compromise between rigorous containment, at the risk of terminating a legitimate process, and less drastic measures. In our embodiment, we have implemented three basic operational modes.

Transparent mode When in this mode, the DDE notifies the CEG about the failed attempts to pull down some files from an external server. The CEG then waits for the file to be actually downloaded, and verifies it is an executable. If so, the CEG crafts a cuckoo's egg with the original file appended. Once the execution of the cuckoo's egg is over, the original file is automatically executed. The cuckoo's egg sends back to the MAE a copy of the parent executing it for analysis. No further action is possible on the suspicious host, as the cuckoo's egg releases the parent process' executable. This mode does not interfere with regular operations of the suspicious host, as the original requested file is executed.

Semi-transparent mode This mode differs from the transparent mode as follows. The original file is downloaded and attached to the cuckoo's egg. However, when the cuckoo's egg is executed, it freezes the parent process. Then, the cuckoo's egg runs the heuristics checks and might decides to "release" its parent process immediately. If the heuristics checks cannot clearly determine the nature of the parent process, the cuckoo's egg ships a copy of the parent process' executable to the MAE. Then, it waits for further commands from the MAE. Further commands may include the termination or release of the process. This mode might interfere with the regular operations of the suspicious host, as the parent process is frozen while the analysis is in progress.

Non-transparent mode When in this mode, the CEG is notified about the failed downloads, but, provided the requested filename points to an executable, does not wait for the original file to be successfully downloaded. Instead, it immediately ships a cuckoo's egg. Based on the heuristic checks, the cuckoo's egg might send back to the MAE a copy of parent executable, and waits for further commands. This mode heavily interfere with the regular host operations, as the requested file is not executed.

2.5 Implementation

To carry out our experiments we have implemented a proof of concept version of Avatar. The three main components of Avatar can be placed at different locations on the network. However, in our experiments we have coupled the DDE and the CEG together into a single host. The reason for this is that the DDE and CEG must exchange information about failed downloads, and the CEG must craft and supply the cuckoo's egg in a timely manner. The deployment of these two components on physically separated systems might introduce delays that could impact the analysis.

In practice, to allow a transparent deployment that does not require any reconfiguration at host side, we employed a single Linux box with built-in firewall and web proxy. The firewall transparently redirects the outgoing traffic directed to common HTTP ports (TCP ports 80 and 8080) through the web proxy, which can inspect both request and reply. Thus, no re-configuration of client hosts is required. As firewall, we use Netfilter, the Linux sub-component in charge of managing network communications. Netfilter offers the possibility to insert specific "hooks" in its packet process workflow, so that it is possible to inspect, and even modify, on-the-fly any packet passing by. To inspect of HTTP traffic, we set up a web proxy based on Apache. Apache supports modules for adding new functionalities, and we have developed a new module to inspect requests and their content. Internally, the module maintains a table that contains statistics about internal hosts and their connection/request failure rates. The module also inspects the replies sent back by the remote (web) server.

When the same host performs several failed connections in a given timeframe, or requests to pull down some file(s) do not end successfully, the Apache module marks that host as suspicious. Depending on the operational mode, the module will either wait until a request is successful, and then ship back a crafted cuckoo's egg together with the original file, or it will immediately ship back a cuckoo's egg (provided the request points to an executable filename).

If the requested file is eventually downloaded, the module proceeds with some sanity checks and verifies that the downloaded file is actually an executable. In case of a positive match, the executable is stored and the cuckoo's egg crafted. To craft the cuckoo's egg and append the requested file, we implement the first method presented in Section 2.2.2 (PE header injection), to avoid download latency.

The analysis engine is implemented as a Windows kernel driver. In order to monitor malware activities, the driver hooks some APIs functions, and exploits the capabilities offered by the latest Windows OSes, which provide built-in sub-systems for third-parties antivirus and firewall software. These interfaces allow one to detect changes in the file system, the system registry, monitor network connections, etc.

Technically speaking, the MAE resides on a real system behind a firewall, in order to prevent any outgoing connection that could be initiated by the malware once it is activated. The MAE does not run on any virtualized environment, to avoid possible built-in anti-analysis capabilities inside the malware. This choice has the disadvantage of requiring a roll back to the original status after each analysis. We do not see this as a serious limitation because our current goal is not to speed up malware analysis, which would require several concurrent systems. Nevertheless, the kernel driver can be deployed in a virtualized environment too.

The cuckoo's egg communicates with the analysis engine through encrypted network sockets. Encryption is used to avoid leaks of any possible sensible information, e.g., a memory dump, over the network, and to prevent the spore from tapping our communications.

2.6 Limitations and evasion of Avatar

In this section we discuss the limitations of our approach.

Limitations of the CEG When crafting the cuckoo's egg, the original requested file can be attached to it. This process could break self-extracting archives, which verify the file integrity before inflating the content.

Evading the DDE Our approach works by first detecting (failed) attempts to download additional components. If malware evades this detection phase, then Avatar cannot ship the cuckoo's egg. To avoid detection, malware could initiate connections at a very low rate, as part of our detection relies on high rate of failed connections. Encrypting connections could be also a countermeasure against inspection.

Evading the CEG Another possible way of evading Avatar is by using some sort of verification mechanism of the downloaded components. Encryption and hashing could be employed to detect a mismatch with the expected file. For instance, by compressing the executable and protecting the archive with a password. Because the sanity check performed on the downloaded file can be solely based on the magic numbers only, a malware writer could hide the executable within a different file type and change the file header at run-time, once downloaded.

Evading the cuckoo's egg Because the cuckoo's egg employs heuristics to decide whether to continue the analysis or to send back to the instrumented host the parent executable for analysis, malware could take some countermeasures to evade the heuristics checks. For instance, since Windows 2000, a process can execute instructions within the context of another process by using the *CreateRemoteThread* API function (a similar function allows the injection of DLLs). Thus, malware could inject arbitrary malicious instructions in the context of an accessible whitelisted process, e.g., Internet Explorer, which is usually executed with the same access rights the malware has, to evade some checks performed by the cuckoo's egg[3].

[3]It is worth noting that the very same technique could be used to evade approaches like the one presented in [13], which relies on the fact that some processes can be whitelisted before hand to avoid false alerts.

Possible Solutions Although we acknowledge that it is possible to devise malware with anti-analysis features tailored for our approach, we did not observe any of those during our experiments. Moreover, the use of encryption of hashing for file verification would likely slow down the malware spread, as either "updated" versions would fail the check or researchers could reverse engineer some malware samples and identify the encryption key/password of the mechanism.

By the way, we think that malware writers might be reluctant in adding a verification step to the malware, as it might simplify the work of signature-based detection system. In the moment that the malware is analyzed the key used for encryption would certainly be identified, and this could be used to craft an effective signature for detecting it.

A possible solution to the evasion of the cuckoo's egg would be to add a comparison of the executable on the disk with the memory image, and pinpoint possible later-added instructions. However, this would require also to inspect DLLs, and the task could easily become infeasible (let alone not being bullet-proof). We plan to address in future work this issue.

3 Benchmarks

To validate the effectiveness of our approach, we use two different datasets. The first data set, referred to as DS_A is available on request from the team that built Malheur. It contains a large collection of malware samples that could be used for malicious purposes. In practice, the data set is a collection of samples submitted in a period of eight consecutive days in 2009. Each sample has been analyzed by CWSandbox and the related report is included together with the original sample. This data set is used to test the basic idea of our approach, that malware will execute an arbitrary generated "egg".

Our second dataset, DS_B, is a collection of malware samples found in the wild. For some samples, no report was available beforehand (meaning they were brand new or modification of known malware samples). Hence, we had to submit the sample to either Anubis or CWSandbox to learn whether the sample was actually malware and downloaded some extra components. With this data set we want to test in particular the effectiveness of the devised heuristics for triggering instrumented analysis of the suspicious process.

3.1 Tests with DS_A

This dataset is an extensive collection of malware samples. They belong to different malware families and are all unique, meaning that some sort of polymorphism/code reordering has been applied.

However, not every sample downloads extra components, and among those which perform download activities, a large part cannot work properly these days. This is due to the fact that, before downloading the extra executables, the malware sample attempts to download some configuration files, which are not longer available. We select only working samples that download additional components, and up to 10 maximum samples per family (in total 75 samples).

To perform the experiment, we set up a client host running Windows XP SP3, as some malware samples suddenly crash when executed under more recent OSes[4], like Windows 7. No extra user activity is simulated. For the DDE, we use the following settings: 5 failed connection/download attempts in 1 minute indicate a possible malicious program. The operational mode for this dataset is set to transparent mode. Table 1 summarizes our findings.

Discussion Tests on DS_A show the effectiveness of our approach. However, we have observed that for few samples and for a certain malware family in particular, the cuckoo's egg is not actually executed. There are two distinct reasons for it. In the case of random samples, once the cuckoo's egg injects its crafted DLL the parent process crashes. In the case of the "Killav" malware family, the malware sample relies on the user to actually execute the download file(s). In all the other cases, there is no check run by the malware whether the downloaded file is actually a "legitimate" malicious component. This enforce our assumption that malware

[4]We investigated this issue and found some incompatibles among installed and expected system libraries.

Malware family	**# of samples**	**# of samples marked as malicious by the DDE**	**# samples that executed the cuckoo's egg**
Agent	9	9	9
Adload	8	6	6
Banload	3	2	2
Chifrax	2	2	2
FraudLoad	8	5	4
Genome	4	4	4
Geral	9	8	8
Killav	6	5	0*
Krap	6	4	4
NothingFound	10	10	3
Xorer	7	6	4

Table 1: Actual samples used in our tests with dataset DS_A, samples flagged as malicious by the DDE and that executed the shipped cuckoo's egg. The * marks a family of malware that actually downloads the cuckoo's egg, but does not automatically execute it (and leaves this to the user). In most cases, the DDE detects failed download attempts, and the cuckoo's egg is executed right away by the malicious sample, without any integrity check.

writers do not currently protect their programs with encryption/hashing mechanisms.

For the "NothingFound" family, whose name might refer to the fact that the submitted sample has not beed identified as malicious by CWSandbox, we have to report that the cuckoo's egg has been actually executed most times.

3.2 Test with DS_B

This dataset is used to tests how our approach performs with (supposedly) brand new malware. Samples have been collected in March 2011, and most of them would have not been detected by several antivirus software at the time of collection (we processed each sample through the VirusTotal [23] web site). We have a total of 30 malware samples from this dataset, which downloads extra malware components. For this set of tests, we also simulate regular user activities such as browsing and downloading, with 30 different software, ranging from web browser to crawlers. Because the downloading program might not execute the cuckoo's egg, we automate its execution and set the parent process to be the downloading program. For the DDE, we use the following stricter settings: 3 failed connection/download attempts in 1 minute will indicate a possible malicious program.

To perform the experiment, similarly to the tests with DS_A, we set up a client host running Windows XP SP3. The operational mode for this dataset is set to semi-transparent mode. By doing so, we test at the same time how efficient heuristics are in detecting malware programs. Because some goodware programs that the heuristics might send to the MAE for analysis could rely on the presence of certain system libraries, for this experiment the MAE is running on a mirror copy of the attacked system. When samples are sent to the MAE, we set a maximum amount of waiting time without operation performed of 3 minutes: By doing so we avoid false positives in case of goodware, but might introduce false negatives in case of malware. Table 2 summarizes our findings.

Discussion This second round of tests confirms that even the latest malware code is still "vulnerable" to the injection of our cuckoo's egg. Most samples have been correctly identified by the DDE, and only 2 samples have been missed. These samples have stopped their download attempts just after a

		# of samples
Malware	Correctly identified by the DDE	28/30
	That executed the cuckoo's egg	27/30 (27/28)
	Correctly identified as malware by heuristics	13/30 (13/27)
	Erroneously identified as goodware by heuristics	2/30 (2/27)
	Sent to the MAE for analysis	12/30 (12/27)
Goodware	Erroneously identified by the DDE	10/30
	Correctly identified as goodware by heuristics	6/30 (6/10)
	Erroneously identified as malware by heuristics	2/30 (2/10)
	Sent to the MAE for analysis	2/30 (2/10)

Table 2: Results for tests with dataset DS_B (in the third column we report partial results in brackets). Almost any malicious download attempt has been detected by the DDE, which shipped the cuckoo's egg. The heuristics identified malware samples in almost half cases, and mistakenly flagged as goodware malicious samples only in a couple of cases. The false positive rate for the DDE is around 30%, and around 20% for the heuristics (when considering the cases in which the cuckoo's egg was shipped).

few tries. The DDE also mistakenly detects as malware some regular programs. Actually this was an expected behaviour, as we set strict values for the DDE. Only one program did not execute the shipped cuckoo's egg, due to a crash at the moment of injection. We experienced the same problem for several samples from DS_A, and our investigations show that the malware was not fully compatible with the installed set of libraries, and therefore would have crashed anyway.

The heuristics prove to be an effective way to reduce the workload of the analysis engine (and thus the manual workload for security analysts), when working in combination with the DDE. Less than 50% of analysed malware samples have been sent to the MAE for analysis, and more than 50% of samples have been immediately identified as malware. In the case of non-malware samples, 20% of analysed samples had to be sent to the MAE for further analysis while 60% has been identified as non-malware without further analysis. For each successful execution, the cuckoo's egg would have been allowed to terminate the parent process, effectively containing a possible infection.

3.3 System overhead

Because the Avatar architecture performs an on-the-fly analysis, and eventually detection, the program execution is slowed down by a certain overhead. Depending on the operational mode, this overhead can range between milliseconds and minutes. A typical example of the former case is transparent mode, when the heuristics flag the downloading program as actual goodware. In non-transparent mode, if the heuristics fail to determine the actual nature of the analysed program, the MAE is activated for further inspection (thus, slowing down the overall analysis time). All our benchmarks have been performed on a 1.8Ghz Dual Core machine (target) and 3.0Ghz Quad Core machine (MAE). Table 3 reports overhead values for different scenarios we observed during our benchmarks (overhead refers to the execution time of the cuckoo's egg and, possibly, of the MAE).

4 Related work

In this section we discuss related works. As malware has become a serious security threat, a good deal of work exists that discusses techniques to analyse and detect malicious code.

4.1 Malware Analysis

Sidiroglou and Keromytis [19] present an architecture to detect and capture potential malware infection vectors by using a collection of heterogenous detec-

Dataset	Min overhead	Avg overhead	Max overhead
DS_A	17ms	21ms	30ms
DS_B (malware samples)	19ms	61s	181s
DS_B (goodware samples)	16ms	29s	181s

Table 3: Overhead time values for DS_A and DS_B. When heuristics successfully identify the analysed sample, the overhead can be as low as 16ms. The maximum overhead value depends on the MAE analysis.

tion engines. Engines range from host-based sensors monitoring the behaviour of applications and OSes to honeypots that simulate possible target applications. Each time a potential malware vector, e.g., a byte stream, is detected, it is copied and forwarded to a sandboxed environment, which runs some instances of the applications one wants to protect (e.g., the Apache web server) and a number of tools to verify the potential maliciousness of the input. The authors provide several strategies for fixing, among others, buffer overflow vulnerabilities "on-the-fly". Despite the fact that authors do not provide any implementation of their architecture, there are several similarities with our approach. Once the cuckoo's egg is being executed, the suspicious program is copied and forwarded to a sandboxed environment for dynamic analysis. The main difference lies in the way we inspect the suspicious program, by crafting the cuckoo's egg and sending it together with the original requested file.

Anubis [3] and CWSandbox [24] are two prominent architectures for dynamic malware analysis. In particular, Anubis can aggregate malware samples that present a similar behaviour into "clusters". That is, although samples' diversity is high (Anubis has analyzed more than 1 million of unique malware samples so far), there are nearly 100.000 malware "families".

4.2 Malware Detection

A number of heterogeneous techniques have been presented to detect malware.

Host-based Techniques Host-based techniques were the first to be used to detect and stop malware (think of antivirus software). Their main advantage is that they can detect malware even before it is actually executed. Approaches range from simple byte-pattern matching, which scans a file for known malicious strings or instructions [21], to model checking [12] and compiler verification [5]. Unfortunately, such (static) techniques can be evaded using packers and polymorphism.

In an effort to overcome typical limitations of matching-based approaches, Kolbitsch et al. [13] introduced a new concept of signature based on fine-grained models. Fine-grained models are graphs representing system calls invocation order (and other additional information) to match the characteristic behavior of a given malware program. The model generation is off-loaded onto a dynamic malware analysis tool (i.e., Anubis). This approach allows the detection of unknown malware samples too, provided the "family" has been analyzed before.

Network-based Techniques Regarding specific network-based techniques, several approaches leverage information extracted by analyzing network traffic [8, 9, 11, 16].

BotMiner [8] combines a number of different traffic monitoring tools to extracts network communication patterns and their content. Typical information that BotMiner takes into consideration are vertical and horizontal scans, exploit attempts, DNS queries, downloads of binaries. Then, BotMiner clusters hosts with a similar behavior and attempts to detect botnet nodes. Although network-based approaches could allow, in theory, to perform on-the-fly detection, this is hard to realize because they miss the activity performed by malware on the host.

Techniques based on Data Mining Several researchers address the detection of malware by using data mining techniques, in a effort to detect a higher number of malware samples that are simply a variant of already known samples.

Tabish et al. [22] notice that most of current malware samples that are daily submitted for analysis are not brand new. Commonly, malware writers employ techniques such as repacking to "obfuscate" malware content and thus defeating approaches based on content matching, e.g., antivirus software. The authors devise an approach based on extracting statistical and information-theoretic features from file blocks. A block is a fixed-sized chunk of byte-level contents of a given file. More than 50 distinct features are extracted, and then analyzed using mathematical distance functions that are common in the data mining field (e.g., the Manhattan and Chebyshev distances). The approach gives in general good results, but requires the analysis of several "good" file samples, e.g., executables, PDF documents, etc., to detect malicious files.

5 Conclusion

In this paper we present Avatar, a new lightweight architecture for on-the-fly, signature-less malware analysis, containment and detection for large networks.

Avatar does not require any special setup or software on the infected hosts. This is because the analysis is not done on the allegedly infected host, but it is carried out on a remote system, which communicates with the (allegedly) infected host through the cuckoo's egg. The cuckoo's egg provides also containment functionalities. In fact, Avatar's architecture is completely centralized. This allows one to deploy it in any environment (like a corporate network) where the firewall can be modified to provide the needed facilities for the interception of suspicious downloads and the injection of the cuckoo's egg. Basically, Avatar can be deployed in most work environments with very little effort. An additional advantage of a centralized architecture is that the updates in the analysis engine affect only one machine, as opposed to what happens e.g., with antivirus software, where all hosts have to be updated.

An interesting aspect of Avatar's architecture is that it can avoid some evasion techniques used by malware; as we mentioned before, modern malware can check whether it is running in a sandboxed environment. Since our architecture does not deploy any extra tool, not even at kernel level, before hand, the malware has little way of detecting that it is under analysis.

The detection in Avatar is necessarily based on heuristics, and is thus fallible. This however allows the detection of malware for which there is no signature available yet. On the other hand, since the heuristics-based detection phase is always followed by an analysis phase before proceeding to the containment, the risk of having false positives in the detection phase is heavily mitigated by the fact that if the analysis phases determines that the suspected malware is actually a legitimate program, the cuckoo's egg can simply "release" it and allow it to continue.

Our experiments show that our approach is effective in detecting and containing malware, even unknown malicious code. We believe that Avatar can be the basis of an effective lightweight first line of defense against malware.

References

[1] Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org`.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS '10: Proc. 17th Network and Distributed System Security Symposium*, 2010.

[3] U. Bayer, A. Moser, C. Krugel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[4] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proc. ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44. ACM Press, 2004.

[5] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *S&P '05: Proc. 25th IEEE Sym-*

posium on Security and Privacy, pages 32–46. IEEE Computer Society, 2005.

[6] P. Milani Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying Dormant Functionality in Malware Programs. In *S&P '10: Proc. 31th IEEE Symposium on Security and Privacy*, page TO APPEAR. IEEE Computer Society Press, 2010.

[7] CWSandbox. `http://www.cwsandbox.org`.

[8] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security '08: Proc. 17th Usenix Security Symposium.* USENIX Association, 2008.

[9] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: detecting malware infection through IDS-driven dialog correlation. In *USENIX Security '07: Proc. 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16. USENIX Association, 2007.

[10] J. Jung, V. Paxson, A.W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *S&P '04: Proc. 25th IEEE Symposium on Security and Privacy*, pages 211–225. IEEE Computer Society Press, 2004.

[11] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proc. 13th USENIX Security Symposium*, pages 271–286. USENIX Association, 2004.

[12] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *DIMVA '05: Proc. 2nd International Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, volume 3548 of *LNCS*, pages 174–187. Springer-Verlag, 2005.

[13] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX '09: Proc. 18th Usenix Security Symposium*, 2009.

[14] Malheur: Automatic Analysis of Malware Behavior. `http://www.mlsec.org/malheur`.

[15] Microsoft. Portable Executable and Common Object File Format Specification, 2008. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[16] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *S&P '05: Proc. 25th IEEE Symposium on Security and Privacy*, pages 226–241. IEEE Computer Society, 2005.

[17] Open Letter to RSA Customers. `http://www.rsa.com/node.aspx?id=3872`.

[18] M. Zubair Shafiq, S. Momina Tabish, F. Mirza, and M. Farooq. PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In *RAID '09: Proc. 12th International Symposium on Recent Advances in Intrusion Detection*, pages 121–141. Springer-Verlag, 2009.

[19] S. Sidiroglou and A.D. Keromytis. A Network Worm Vaccine Architecture. In *WETICE '03: Proc. 12th International Workshop on Enabling Technologies*, pages 220–225. IEEE Computer Society, 2003.

[20] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *CCS '09: Proc. 16th ACM conference on Computer and Communications Security*, pages 635–647. ACM Press, 2009.

[21] P. Szor. *The Art of Computer Virus Research and Defense.* Addison-Wesley Professional, 2005.

[22] S. Momina Tabish, M. Zubair Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *CSI-KDD '09:*

Proc. ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, pages 23–31. ACM Press, 2009.

[23] VirusTotal: Online Virus, Malware and URL Scanner. `http://www.virustotal.com`.

[24] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.